MINISTRY OF EDUCATION AND SCIENTIFIC RESEARCH

TECHNICAL UNIVERSITY
OF CLUJ-NAPOCA

FACULTY OF AUTOMATION AND COMPUTER SCIENCE
COMPUTER SCIENCE DEPARTMENT

3-tier REST Services

Hands on SPRING REST Framework

Ioan Salomie　Tudor Cioara　Ionut Anghel
Marcel Antal　Claudia Daniela Pop　Cristina Pop

DSRL Distributed Systems Research Laboratory

Contents

1. Introduction 3
   1.1. Get the project..3
   1.2. Project Structure 5
2. Project Conceptual Architecture..6
3. Project Functionality..8
4. Validation and Error Handling 15
   4.1. Annotation Based Validation..15
   4.2. Error Handling..17
5. Testing the Project 18
6. Deploy on Web Server..19
7. Reinforcement Learning..20
8. References 20

1. Introduction

The Spring Framework is an application framework for the Java platform. The framework was first released on the 1st October 2002 and was written by the Australian computer specialist Rod B. Johnson. Due to its continuous enhancement and development, the framework became widely used in software companies nowadays. The project example from this laboratory work is a skeleton for a Spring application that can be used to get the information associated with the users described in a database.

1.1. Get the project

1. Setup GIT and download the project from https://gitlab.com/ds_20201/spring-demo.git
   - Create an empty local folder in the workspace on your computer
   - Right-click in the folder and select **Git Bash**
   - Write the commands:
     - git clone https://gitlab.com/ds_20201/spring-demo.git

2. Create an empty database in PostgreSQL with the name **city-db**
3. Import the project in IntelliJ
4. Check the **application.properties** file from src/main/resources and fill the database.user and database.password of the local PostgreSQL server.
5. In the **application.properties** file change the ddl-auto flag from validate to create in order for the Spring Application to be able to create the structure of the tables:
   - *spring.jpa.hibernate.ddl-auto = create*

   Observation! Make sure you change the property back to **validate**, in order to avoid recreating the database at the following restart.

6. Run the Application in IntelliJ: Right-click on the class Ds2020Application and select **Run 'Ds2020Application'**. (Figure 1)
7. Go to your workbench application (pdAdmin) and insert a person in the database using the following query

```
INSERT INTO person (id, name, address, age)
VALUES (decode(replace('45774962-e6f7-41f6-b940-72ef63fa1943'::text,
'-', ''), 'hex'), 'Its me', 'My Address', 22);
```

Observation! Once inserted the Person in the database, the UUID will be stored in a binary format, thus the id retrieved should not be copied in the plain form, but should be hex encoded.

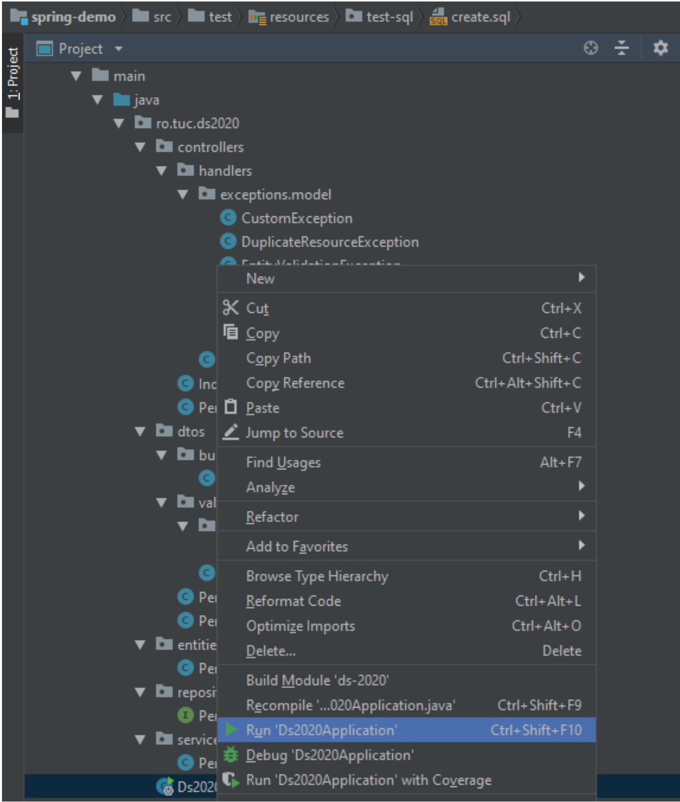

Figure 1 Run Application

8. Test the implemented REST requests:

- http://localhost:8080/ => this request should display the message "City APP Service is running..."
- http://localhost:8080/person => this request should retrieve all the persons from the database
- http://localhost:8080/person/45774962-e6f7-41f6-b940-72ef63fa1943 => this request should retrieve the person with the UUID 45774962-e6f7-41f6-b940-72ef63fa1943

```
{
    "id": "45774962-e6f7-41f6-b940-72ef63fa1943",
    "name": "Its me",
    "age": 22
}
```

Figure 2 Result of the Query that Retrieves a Person by UUID from the Database

9. The project can also be tested from IntelliJ, running the instructions clean and install. Upon successful building, 8 out of the 8 tests should run successfully.

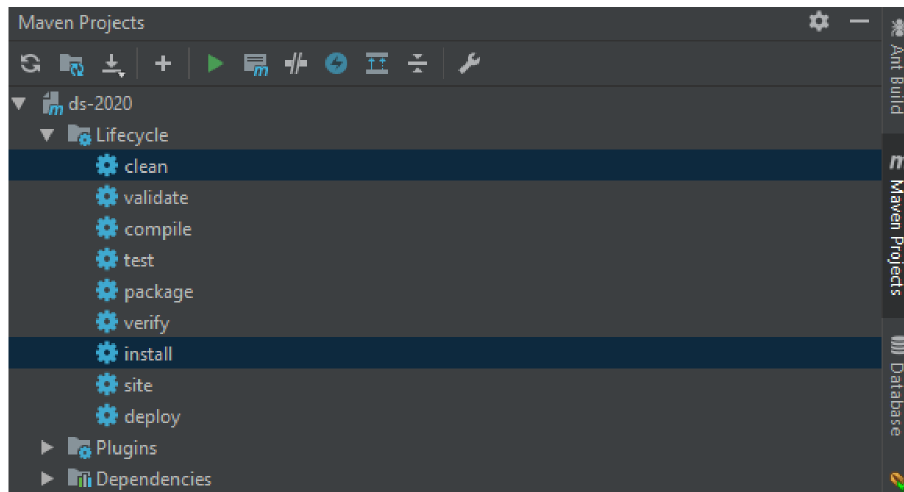

Figure 3 Build the Project in IntelliJ

```
[INFO] Results:
[INFO]
[INFO] Tests run: 8, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO]
[INFO] --- jacoco-maven-plugin:0.8.3:report (generate-code-coverage-report) @ ds-2020 ---
[INFO] Loading execution data file E:\DS2020\spring-demo\target\jacoco.exec
[INFO] Analyzed bundle 'ds-2020' with 16 classes
[INFO]
[INFO] --- jacoco-maven-plugin:0.8.3:report (post-unit-test) @ ds-2020 ---
[INFO] Loading execution data file E:\DS2020\spring-demo\target\jacoco.exec
[INFO] Analyzed bundle 'ds-2020' with 16 classes
[INFO]
```

Figure 4 Test Results

1.2. Project Structure

The project presents some basic operations on the person entity and aims at presenting the layers involved in performing CRUD operations on the person table shown in the following figure:

When opened in IntelliJ, the project has the following structure shown in Figure 7. All the components are detailed in Chapter 2.

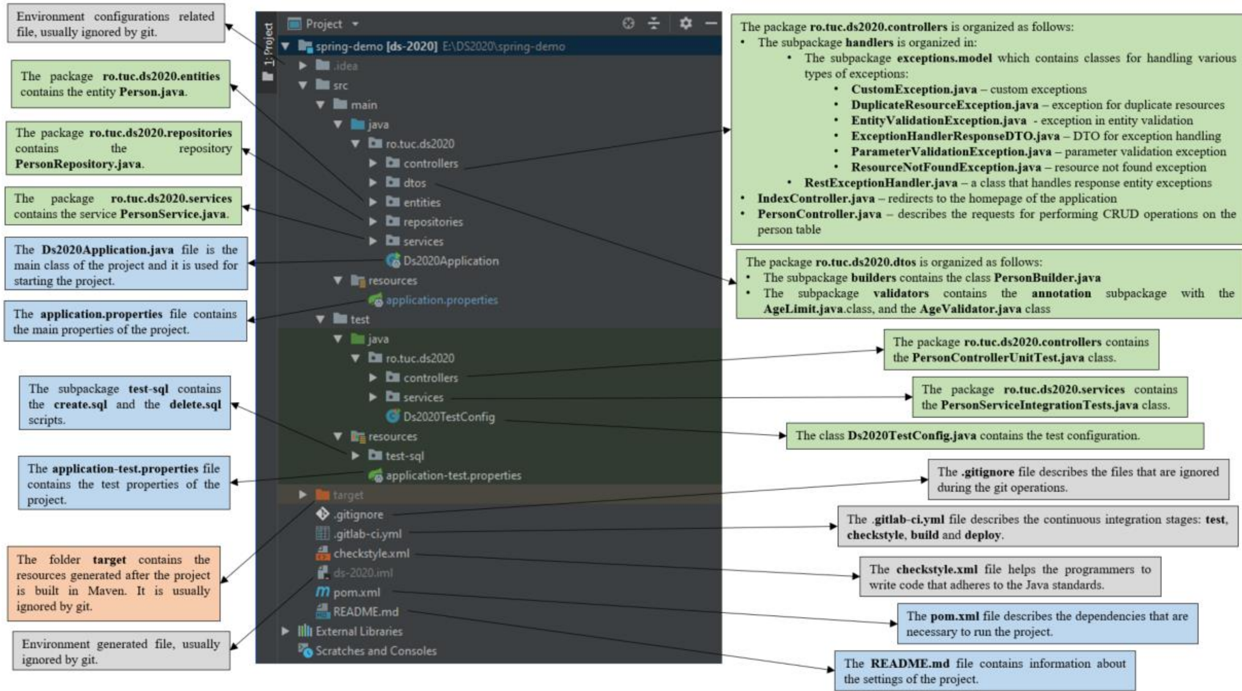

Figure 5 Project Structure in IntelliJ

2. Project Conceptual Architecture

The conceptual architecture of the system is presented below.

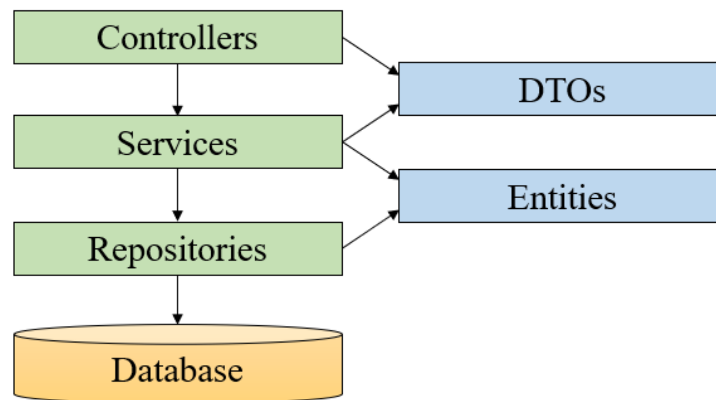

Figure 6 Project Conceptual Architecture

The following table describes each component:

Table 2-1- Project Components Description

Component	Package	Description
Repositories	ro.tuc.ds2020.repositories	Package that contains the repositories, classes that facilitate the DB access. The developer can use custom queries to communicate with the DB.
Entities	ro.tuc.ds2020.entities	An entity corresponds to a table from the relational database and each instance of the entity corresponds to a row from the database
Services	ro.tuc.ds2020.services	This layer represents the business logic layer of the Spring application. It translates the Data Transfer Objects (DTOs) into entities and back. For formatting the values from DTO objects to Entity objects Builder classes are used. The service layer is responsible to apply more complex operations and validations before accessing the repository layer.
DTOs	ro.tuc.ds2020.dtos	A Data Transfer Object (DTO) is a special object exposed outside the application (to the UI or APIs). It contains part of the underlying Entities or combinations of different entities. Additionally, it contains builders and validators.
Controller	ro.tuc.ds2020.controllers	The layer that exposes the application functionality as an API able to handle HTTP REST requests. It also contains handlers for various types of exceptions.

3. Project Functionality

A simple sequence diagram that involves the interactions between the components is shown in Figure 7:

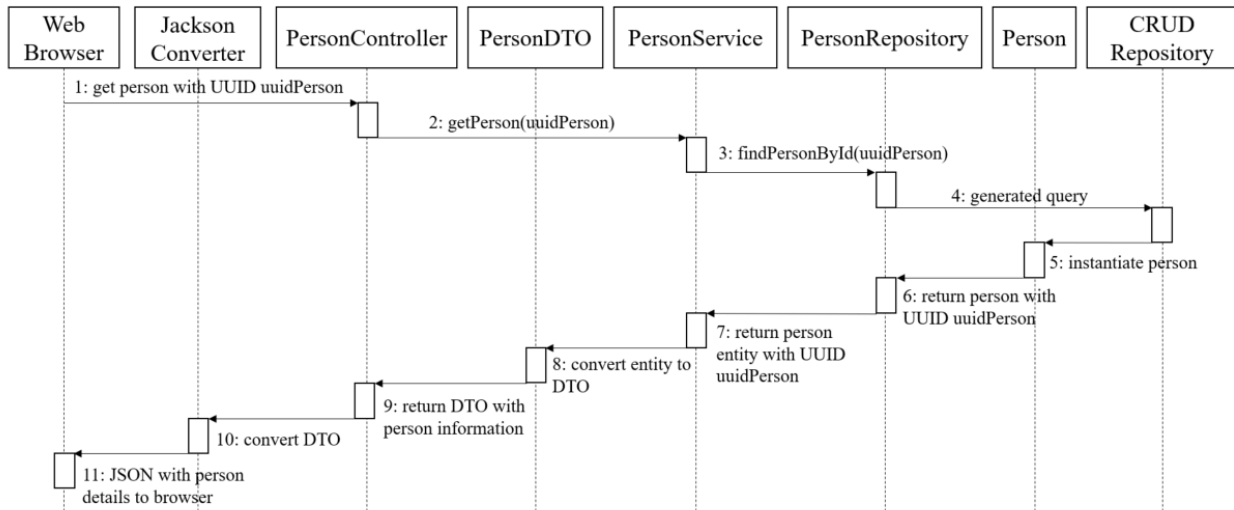

Figure 7 Sequence Diagram for GET operation

The processing steps for a person with the uuidPerson 45774962-e6f7-41f6-b940-72ef63fa1943 are described in the following section:

1. Web Browser sends a HTTP request with the method GET to retrieve the user with UUID = 45774962-e6f7-41f6-b940-72ef63fa1943. This happens by calling the URL: http://localhost:8080/person/45774962-e6f7-41f6-b940-72ef63fa1943. This URL is composed of the following parts:
   - **http**: protocol used to communicate
   - **localhost**: address of the server to communicate with. This can be either an URL resolved by DNS to an IP address, or an IP address. (localhost or 127.0.0.1 in this case).
   - **8080**: the port on which the web server which will respond to the request is listening.
   - **http://localhost:8080/person/45774962-e6f7-41f6-b940-72ef63fa1943** . The last part of the address is mapped to the resources within the application by the web server. In this case, the application exposes a REST API through its **controllers**. The mapping is done in ro.tuc.ds2020.controllers.PersonController.java in three steps, as follows:
     i) mapping to the controller: @RequestMapping(value = "/person") (line 18)
     ii) mapping to the method within the controller and defining the request type: @GetMapping(value = "/{id}") (line 40)
     iii) defining the parameters of the method at (line 40) and (line 41) (the name in the request must correspond to the name within the @PathVariable tag. Inside the method, the Java Parameter is used – **int id**).

2. The Spring controller ro.tuc.ds2020.controllers.PersonController.java has already instantiated a service instance due to the annotation @Autowired (line 23). Using this ro.tuc.ds2020.services.PersonService.java object, inside the getPerson() method it calls the findPersonById() method (line 42), delegating the processing to the service layer.

Good to know

The **Controllers Layer** is a layer over the **Services Layer** and calls the methods which are provided by the **Services Layer**.

a) How are the controllers defined?

The controllers are defined using the annotation @RestController. This annotation specifies the fact that the corresponding annotated class can handle RESTful WEB Services. The REST (Representational State Transfer) services describe one way of communication between different computer systems on the internet. REST uses HTTP (Hyper Text Transfer Protocol) for the communication with the internet resources.

b) How are the controllers mapped to the URLs?

The controllers are mapped to the URLs using mapping annotations. These annotations are used in two cases:

- For annotating the entire class – in this case the value of the @RequestMapping (line 18) is a **prefix** for all the other URLs that are handled by the controller.

```
16  @RestController
17  @CrossOrigin
18  @RequestMapping(value = "/person")
19  public class PersonController {
20
21      private final PersonService personService;
```

Figure 8 RequestMapping for PersonController

- For annotating a specific method – in this case the value of the @GetMapping (line 40) is a **suffix** for the URL that corresponds to the method. The parameter **value** describes the location while the request method is inferred from the name of the annotation. There are various types of mapping annotations such as: @GetMapping, @PostMapping, @PutMapping and @DeleteMapping.

```
28      @GetMapping()
29      public ResponseEntity<List<PersonDTO>> getPersons() {
30          List<PersonDTO> dtos = personService.findPersons();
31          return new ResponseEntity<>(dtos, HttpStatus.OK);
32      }
```

Figure 9 GetMapping for getting all the persons

c) How are the **Services Layer** instances accessed?

- The objects from the **Services Layer** are accessed using the @Autowired annotation.

d) What are the most common input parameters of the methods annotated with @GetMapping or @PostMapping?

- A path variable – in this case the variable is a part of the path specified by the @GetMapping annotation

```java
40    @GetMapping(value = "/{id}")
41    public ResponseEntity<PersonDTO> getPerson(@PathVariable("id") UUID personId) {
42        PersonDTO dto = personService.findPersonById(personId);
43        return new ResponseEntity<>(dto, HttpStatus.OK);
44    }
```

Figure 2 Example that uses the @PathVariable annotation

- A request body – in this case the object is a DTO that contains information to be inserted in the application

```java
34    @PostMapping()
35    public ResponseEntity<UUID> insertPerson(@Valid @RequestBody PersonDetailsDTO personDTO) {
36        UUID personID = personService.insert(personDTO);
37        return new ResponseEntity<>(personID, HttpStatus.CREATED);
38    }
```

Figure 3 Example that uses the @RequestBody annotation

3. The PersonService object is called with the method findPersonById(). It uses the personRepository object that was injected due to the @Autowired annotation (line 24) to find the person in the DB.

Good to know

The **Services Layer** is an intermediary layer between the **Repositories Layer** and the **Controllers Layer**.

a) What are the services?

The services provide transactional operations for the business logic. A service method either completes or the database rollbacks to the previous state.

b) What is the purpose of the **Services Layer**?

The purpose of the **Services Layer** is to define methods that perform several operations on a database in such a way that either all the operations execute successfully or none of them is executed. In the second case the database rollbacks to the original state. It is responsible to process the data before accessing the repository layer, in order to ensure that all the business logic rules hold true.

c) How are the services defined?

The services are defined using the annotation @Component or the annotation @Service.

d) How are the objects defined in the **Repositories Layer** accessed?

The objects from the **Repositories Layer** are accessed using the annotation @Autowired (line 24). The purpose of this annotation is to auto wire the Spring beans.

```java
@Service
public class PersonService {
    private static final Logger LOGGER = LoggerFactory.getLogger(PersonService.class);
    private final PersonRepository personRepository;

    @Autowired
    public PersonService(PersonRepository personRepository) { this.personRepository = personRepository; }

    public List<PersonDTO> findPersons() {
        List<Person> personList = personRepository.findAll();
        return personList.stream()
                .map(PersonBuilder::toPersonDTO)
                .collect(Collectors.toList());
    }
```

Figure 4 PersonService

e) Why does the Services Layer use DTOs instead of entities?

Usually, the DTOs reduce the overhead between the backend and the presentation. The optimized DTOs contain only that information which is absolutely required. Furthermore, the purpose of the DTO is also to restrict the access to the information exposed by the entities. Depending on the roles/ rights of the requesting clients, some information may not be allowed to be exposed, thus DTOs can be used to restructure the information exposed on the external services.

4. The PersonRepository method findById is called. This method uses an auto generated query by the JpaRepository<Person, UUID> superclass to retrieve a user by its id.

Good to know

The **Repositories Layer** intermediates the communication between the **Services Layer** and the **Database**.

a) How are the repositories defined?

The repositories are defined by extending the interface **JpaRepository<T, ID extends Serializable>**. The first argument **T** describes the type of the entities used by the repositories while the second argument **ID** describes the type of the id of the entities.

```java
public interface PersonRepository extends JpaRepository<Person, UUID> {

    /**
     * Example: JPA generate Query by Field
     */
    List<Person> findByName(String name);

    /**
     * Example: Write Custom Query
     */
    @Query(value = "SELECT p " +
            "FROM Person p " +
            "WHERE p.name = :name " +
            "AND p.age >= 60  ")
    Optional<Person> findSeniorsByName(@Param("name") String name);

}
```

Figure 5 PersonRepository interface

The **PersonRepository** handles entities of the type **Person** which have the id of the type **UUID**.

b) How to access the database using the Spring repositories?

There are different ways to access the database:

- Use one of the methods declared by the **JpaRepository**. The CRUD operations are implemented by default by the **JpaRepository** and it is not necessary to declare them again in the interface that extends it

```
7   @org.springframework.data.repository.NoRepositoryBean
8   public interface JpaRepository <T, ID> extends org.springframework.data.repository.
9       java.util.List<T> findAll();
```

Figure 6 JpaRepository snippet

- Create methods based on the fields from the entity (e.g. findBy**Id**, findBy**Name**, etc.). In this case the name of the method is parsed and interpreted by the Spring framework in order to execute the corresponding query. Also, there is the possibility to create queries which are more complex with filters, join and so on (for more details please see the **JpaRepostiory** documentation)

```
14      /**
15       * Example: JPA generate Query by Field
16       */
17      List<Person> findByName(String name);
```

Figure 7 UserRepository interface

- Use custom defined queries – in the case of the custom defined queries the name of the method is not parsed; the purpose of the **@Param** annotation is to specify the names of the parameters which are used in the definition of the query

```
19      /**
20       * Example: Write Custom Query
21       */
22      @Query(value = "SELECT p " +
23              "FROM Person p " +
24              "WHERE p.name = :name " +
25              "AND p.age >= 60 ")
26      Optional<Person> findSeniorsByName(@Param("name") String name);
```

Figure 8 Custom Defined Queries

5. The UserRepository retrieves a user entity object instantiated with values from the DB.

Good to know
a) What are the entities? An entity represents a table from the relational database and each instance of the entity corresponds to a row from the database. An example of entity is shown in Figure 17.

```java
@Entity
public class Person  implements Serializable{

    private static final long serialVersionUID = 1L;

    @Id
    @GeneratedValue(generator = "uuid2")
    @GenericGenerator(name = "uuid2", strategy = "uuid2")
    @Type(type = "uuid-binary")
    private UUID id;

    @Column(name = "name", nullable = false)
    private String name;

    @Column(name = "address", nullable = false)
    private String address;

    @Column(name = "age", nullable = false)
    private int age;
```

Figure 9 Person Entity

b) What are the main requirements for the creation of the entities?

- The entity class must be annotated with the annotation @Entity
- The id and the columns are mapped using the annotations @Id and @Column
- The class **must** have one public/protected no-argument constructor

c) Which are the most common annotations used by the entities?

The most common annotations which are used in the mapping process are described below:

- @Entity – specifies the fact that the class which is annotated with this annotation is an entity
- @Table – specifies the table to which the entity is mapped
- @Id – the annotated field is an ID of the table
- @Column – the annotated fields are columns of the table from the database
- @OneToOne – maps the one-to-one relationship between two tables
- @OneToMany – maps the one-to-many relationship between two tables
- @ManyToOne – maps the many-to-one relationship between two tables
- @ManyToMany – maps the many-to-many relationship between two tables

6. A person entity is returned to the PersonRepository.
7. A person DTO is returned to the PersonService.
8. The PersonService converts the entity object to a Data Transfer Object (DTO). The **BUILDER** Design Pattern is used in this case to ease the adaptation between the two classes.

Good to know

a) What are the DTOs?

The DTOs (Data Transfer Objects) are objects that carry data between processes and are exposed by the application to the UI or through an API.

b) What is the relation between the DTOs and the entities?

If the database changes then the mappings used by the entities must also change, but the objects (DTOs) might remain unchanged.

c) Why are the DTOs used?

The motivation for using the DTOs is represented by the fact that they reduce the cost of communication between the processes. The DTOs aggregate in one call data that might be transferred by several calls. Furthermore, DTO provide the option to selectively hide sensitive data, that otherwise would be exposed if using directly the entity object.

d) How to convert between entities and DTOs?

One possibility to convert the entities to DTOs or vice-versa is to use a class that contains static methods as in the figure presented below.

```
public class PersonBuilder {

    private PersonBuilder() {
    }

    public static PersonDTO toPersonDTO(Person person) {
        return new PersonDTO(person.getId(), person.getName(), person.getAge());
    }

    public static Person toEntity(PersonDetailsDTO personDetailsDTO) {
        return new Person(personDetailsDTO.getName(),
                personDetailsDTO.getAddress(),
                personDetailsDTO.getAge());
    }
}
```

Figure 1810 PersonBuilder

9. A PersonDTO is returned to the controller.
10. Transparent to the programmer, the Spring framework calls the Jackson Converter to convert the retrieved user DTO to a JSON object that will be sent to the browser.

```
{
"id": "45774962-e6f7-41f6-b940-72ef63fa1943",
"name": "Its me",
"age": 22
}
```

Figure 19 JSON with data from the DTO object

11. The Data is displayed in the browser:

4. Validation and Error Handling

There are several options possible for validating the data received from a HTTP client, that aims to insert/update different resources managed by the server application. The most common one is to manually validate the data in code using different conditionals to check if certain fields are null, empty, etc. However, a more optimal approach is to use Annotation Based validation. For this there are several things you need to consider.

4.1. Annotation Based Validation

1. Add the necessary validation **dependencies** in pom.xml

```
<dependency>
    <groupId>javax.validation</groupId>
    <artifactId>validation-api</artifactId>
    <version>2.0.0.Final</version>
</dependency>
<dependency>
    <groupId>org.hibernate.validator</groupId>
    <artifactId>hibernate-validator</artifactId>
    <version>6.0.2.Final</version>
</dependency>
<dependency>
    <groupId>org.hibernate.validator</groupId>
    <artifactId>hibernate-validator-annotation-processor</artifactId>
    <version>6.0.2.Final</version>
</dependency>
```

Figure 20 Validation Dependencies

2. Annotate the startup class (Ds2020Application) with the **@Validated annotation**

```
@SpringBootApplication
@Validated
public class Ds2020Application extends SpringBootServletInitializer {
```

Figure 21 Enable Validation at application level

3. Mark the parameters that need to be validated by using the **@Valid annotation** in the Controller's methods:

```
@PostMapping()
public ResponseEntity<UUID> insertProsumer(@Valid @RequestBody PersonDetailsDTO personDTO) {
    UUID personID = personService.insert(personDTO);
    return new ResponseEntity<>(personID, HttpStatus.CREATED);
}
```

Figure 22 Apply Validation at method parameters level

   Once the request gets in the controller, the request body object (personDTO) will automatically be validated according to the **validation rules**.

4. The **validation rules** are specified in the PersonDetailsDTO class through annotations:

```java
public class PersonDetailsDTO {

    private UUID id;
    @NotNull
    private String name;
    @NotNull
    private String address;
    @AgeLimit(limit = 18)
    private int age;
```

Figure 23 Configure Validation Rules

There are predefined annotations, like @NotNull, however you can also define your own custom annotations that should be used as validator.

5. @AgeLimit is defined as a custom annotation, that given the age limit will validate whether the value of the age field is greater than the limit imposed through the annotation (*limit=18*).

The *AgeLimit* annotation is defined in ro.tuc.ds2020.dtos.validators.annotation package.

```java
9   @Target({ElementType.METHOD, ElementType.FIELD, ElementType.PARAMETER})
10  @Retention(RetentionPolicy.RUNTIME)
11  @Documented
12  @Constraint(validatedBy = {AgeValidator.class})
13  public @interface AgeLimit {
14
15      int limit() default 120;
16
17      String message() default "Age  does not match the required adult limit";
18
19      Class<?>[] groups() default {};
20
21      Class<? extends Payload>[] payload() default {};
22
23  }
```

Figure 24 AgeLimit Annotation

Here you can define the error message (line 17), default values for the limit (line 15) and most important the Validator class that provides the actual validation code (line 12).

```java
@Component
public class AgeValidator implements ConstraintValidator<AgeLimit, Integer> {

    private int ageLimit;

    @Override
    public void initialize(AgeLimit constraintAnnotation) { this.ageLimit = constraintAnnotation.limit(); }

    @Override
    public boolean isValid(Integer inputAge, ConstraintValidatorContext constraintValidatorContext) {
        return inputAge > ageLimit;
    }
```

Figure 25 AgeLimit Validator

The *AgeValidator* class is defined in the ro.tuc.ds2020.dtos.validators package. By implementing the *ConstraingValidator* interface*,* the annotation that the validator is responsible for and the input types are specified. During the *initialize* method the age provided as a limit through the annotation is loaded and the *isValid* method validates the input provided through the age field against the *ageLimit* set through the annotation.

4.2. Error Handling

Whenever an exception is encountered in the application (can be due to a validation constraint that failed, or a custom thrown exception), a *RestExceptionHandler* is defined to gather all these exceptions and handle them in a uniform approach.
The Handler class is annotated with the *@ControllerAdvice* annotation and can be found in the ro.tuc.ds2020.controllers.handlers package.

```
20  @ControllerAdvice
21  public class RestExceptionHandler extends ResponseEntityExceptionHandler {
22
23      @Override
24      protected ResponseEntity<Object> handleMethodArgumentNotValid(MethodArgumentNotValidException ex,
25                                                                    HttpHeaders headers,
26                                                                    HttpStatus status,
27                                                                    WebRequest request) {...}
49
50      @ExceptionHandler(value = {ResourceNotFoundException.class, DuplicateResourceException.class, EntityValidationException.class})
51      protected ResponseEntity<Object> handleResourceNotFound(CustomException ex,
52                                                              WebRequest request) {...}
```

Figure 26 Application Level Exception Handler

As noticed in the previous figure the class is responsible to handle exception as (line 50): ResurceNotFound, DuplicateEntry and EntityValidation. These are custom exceptions defined in ro.tuc.ds2020.controllers.handlers.exceptions.model package. These are exceptions that can be thrown from the service layer as depicted in the following figure, line 40:

```
36      public PersonDTO findPersonById(UUID id) {
37          Optional<Person> prosumerOptional = personRepository.findById(id);
38          if (!prosumerOptional.isPresent()) {
39              LOGGER.error("Person with id {} was not found in db", id);
40              throw new ResourceNotFoundException(Person.class.getSimpleName() + " with id: " + id);
41          }
42          return PersonBuilder.toPersonDTO(prosumerOptional.get());
43      }
```

Figure 27 Resource Not Found exception thrown

Furthermore, the validation exception thrown because of violating the constraints presented in section 4.1, are handled through *handleMethodArgumentNotValid* at line 24.

As a result, when a POST is issued with invalid data, providing a minor person, the message provided in the custom annotation will be returned and the 400 HTTP code set on the result.

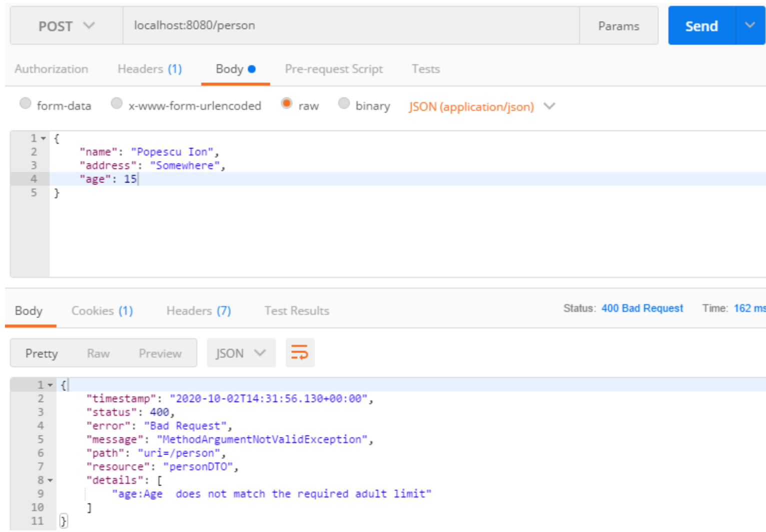

Figure 28 Validation Failed POSTMAN Example

Similarly, when requesting a person that does not exist in the database, a 404 HTTP code is returned, together with a JSON in the response body specifying the resource and the cause together with other details about the request:

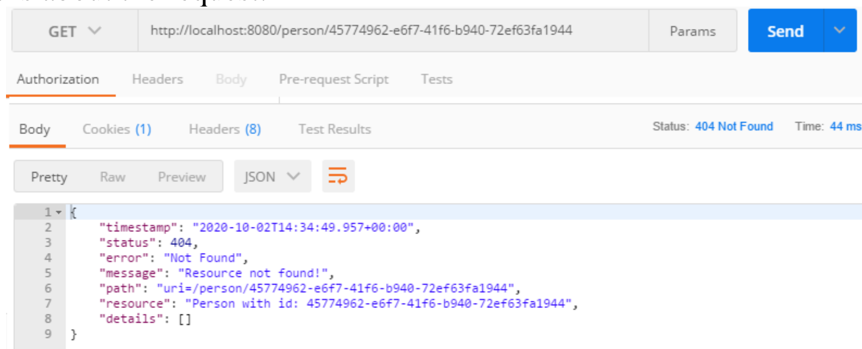

Figure 29 Resource Not Found POSTMAN Example

5. Testing the Project

1. Configure the database properties from the **src/main/resources/application.properties** file
2. Run the project as: Run Click on Ds2020Application > Run 'Ds2020Application'
3. Use a tool such as POSTMAN to make REST API calls and retrieve or insert persons from/in the database:

   URL: localhost:8080/person/

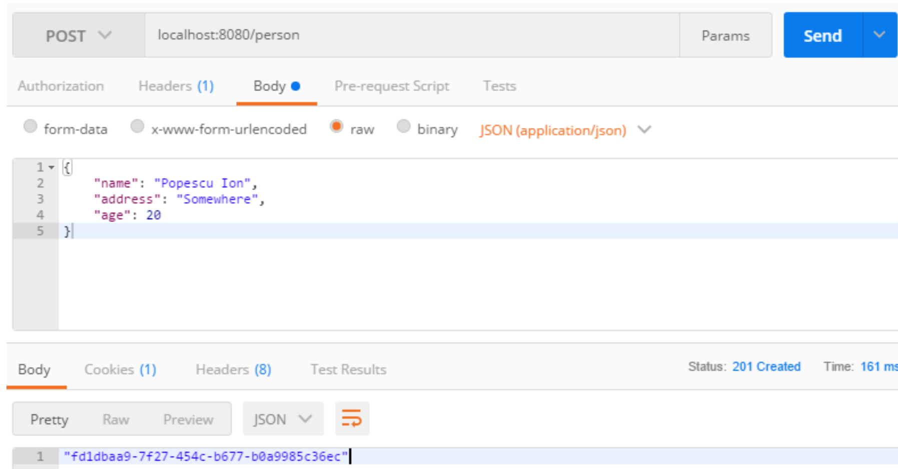

Figure 30 Example of using Postman tool

6. Deploy on Web Server

Deploy the Spring application on a Tomcat Web Server:

- Modify the packaging in pom.xml from JAR to WAR

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.(
    <modelVersion>4.0.0</modelVersion>
    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>2.3.3.RELEASE</version>
        <relativePath/> <!-- lookup parent from repository -->
    </parent>
    <groupId>ro.tuc</groupId>
    <artifactId>ds-2020</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <name>ds-2020</name>
    <description>Demo project for Spring Boot</description>
    <packaging>jar</packaging>
```

- Build the project (in IntelliJ: Maven-> Clean -> Install)
- Copy the WAR from **project/target** to **/apache-tomcat/webapps**
- Run Apache Tomcat: **/bin/startup.bat**

7. Reinforcement Learning

Answer the following questions:

- What is the Inversion of Control (IoC)?
- What is the Dependency Injection?
- Explain the @Autowired annotation.
- Explain the @Entity annotation.
- What is the Spring IoC container?
- What are the Spring beans?

8. References

[1] http://docs.spring.io/spring/docs/current/spring-framework-reference/html/overview.html
[2] https://en.wikipedia.org/wiki/AspectJ
[3] https://docs.spring.io/spring-boot/docs/current/reference/html/spring-boot-features.html#boot-features-messaging
[4] http://www.baeldung.com/websockets-spring
[5] http://docs.oracle.com/javaee/5/tutorial/doc/bnbqa.html